Generador de Abstracciones de Comportamiento para Contratos Inteligentes mediante Fuzzing

Ian Grinspan, Javier Godoy, and Diego Garbervetsky

Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Ciudad Autónoma de Buenos Aires, Argentina
{igrinspan, jgodoy, diegog}@dc.uba.ar

Resumen Los contratos inteligentes son programas inmutables en la blockchain que gestionan activos digitales sin intermediarios. Su inmutabilidad y manejo de recursos valiosos hacen crucial la detección temprana de errores. Este trabajo presenta un enfoque para generar automáticamente abstracciones por predicados en contratos Solidity de la red Ethereum, utilizando análisis dinámico vía fuzzing con la herramienta de código abierto Echidna. Se desarrolla un prototipo que explora el comportamiento real de los contratos, generando máquinas de estado finitas que abstraen el comportamiento de los mismos, basándose en predicados que reflejan el estado del contrato y las precondiciones necesarias para la habilitación de sus funciones. Esto permite identificar comportamientos complejos difíciles de detectar manualmente. Finalmente, se comparan las ventajas y limitaciones del enfoque en relación al uso de herramientas de análisis estático.

Keywords: Validación de Software, Análisis Dinámico, Fuzzing, Blockchain, Abstracciones por Predicados

Smart Contract Behavior Abstraction Generator using Fuzzing

Abstract Smart contracts are immutable programs on the blockchain that manage digital assets without intermediaries. Their immutability and handling of valuable resources make early error detection crucial. This work presents an approach to automatically generate predicate abstractions in Solidity contracts on the Ethereum network, using dynamic analysis through fuzzing with the open-source tool Echidna. A prototype is developed to explore the actual behavior of contracts, generating finite state machines that abstract their behavior based on predicates that reflect the contract's state and the preconditions required to enable its functions. This allows the identification of complex behaviors that are difficult to detect manually. Finally, the advantages and limitations of the approach are compared to the use of static analysis tools.

Keywords: Software Validation, Dynamic Analysis, Fuzzing, Blockchain, Predicate Abstractions

1. Introducción

El presente trabajo se enfoca en el análisis del comportamiento de contratos inteligentes escritos en Solidity para la red de Ethereum, mediante la construcción de abstracciones que modelan su funcionamiento. Con el crecimiento exponencial del uso de la tecnología blockchain, los contratos inteligentes han adquirido un rol fundamental en el uso de activos digitales y la creación de aplicaciones descentralizadas, entre otras cosas. Sin embargo, la complejidad de estos programas, sumada a su inmutabilidad, plantea desafíos significativos en términos de seguridad. Detectar errores o vulnerabilidades en los contratos antes de su despliegue es crítico para prevenir pérdidas económicas.

Este contexto ha dado lugar al desarrollo de herramientas que buscan analizar contratos inteligentes mediante distintas técnicas, principalmente de análisis estático mediante verificación formal y model checking. Estas metodologías intentan garantizar que los contratos cumplan con propiedades específicas mediante un análisis exhaustivo de todas sus posibles ejecuciones. Sin embargo, generalmente requieren de intervención humana en alguna de sus etapas. Es común complementar el proceso con auditorías de seguridad manuales, en las que expertos revisan el código en busca de vulnerabilidades que podrían pasar desapercibidas en un análisis automatizado. A la vez, existen herramientas que buscan crear modelos abstractos que capturen el comportamiento de los contratos, facilitando su interpretación y evaluación tanto para desarrolladores como para auditores. No obstante, cada enfoque presenta limitaciones que motivan la búsqueda de otro tipo de técnicas.

La principal contribución de este trabajo es el desarrollo de una herramienta que, a partir del código fuente de contratos inteligentes escritos en Solidity, genera automáticamente abstracciones por predicado realizando un análisis dinámico mediante campañas de fuzzing. Esta técnica permite explorar estados y transiciones alcanzables de un contrato, proporcionando una visión general de su comportamiento y facilitando la detección de errores o inconsistencias.

Para validar la efectividad de la herramienta, se realizaron experimentos sobre un conjunto de benchmarks. Se compararon las abstracciones obtenidas con las ya existentes para dichos benchmarks y se analizó el impacto de diversas configuraciones y optimizaciones en la ejecución. En particular, se busca responder preguntas clave, como la viabilidad de generar abstracciones dinámicas a partir de contratos inteligentes y la utilidad práctica de dichas abstracciones para detectar errores. Además, se compararon los resultados utilizando el mismo procedimiento con un enfoque basado en herramientas de análisis estático, introducido en trabajos previos.

En síntesis, este trabajo busca aportar una nueva perspectiva al análisis de contratos inteligentes, combinando técnicas de fuzzing y abstracción de estados con herramientas existentes y proponiendo un enfoque que puede sentar las bases para investigaciones futuras.

2. Preliminares

2.1. Contratos Inteligentes

Una blockchain[1] es una estructura de datos que permite almacenar información de forma distribuida, segura y públicamente verificable. Ethereum[2], una de las plataformas más utilizadas, extiende esta tecnología mediante la ejecución de programas llamados contratos inteligentes.

A diferencia del modelo basado en transacciones de Bitcoin[3], Ethereum utiliza un modelo basado en cuentas, lo que permite mantener un estado global que se actualiza con cada transacción. Este estado incluye el código y almacenamiento asociado a cada contrato inteligente desplegado en la red.

Los contratos inteligentes se ejecutan de forma determinista en la Ethereum Virtual Machine (EVM)[4] y están escritos comúnmente en Solidity[5][6], un lenguaje de alto nivel diseñado específicamente para esta plataforma, compilado por el compilador solc[7] y traducido a bytecode listo para ejecutar por la EVM. Estos contratos permiten automatizar reglas, manejar activos digitales y construir aplicaciones descentralizadas.

Una propiedad fundamental de los contratos inteligentes es su inmutabilidad: una vez desplegados en la blockchain, su código no puede ser modificado. Esto implica que cualquier error o vulnerabilidad permanecerá indefinidamente, con un alto riesgo de pérdidas de activos. Por esta razón, es imprescindible realizar pruebas rigurosas antes de su despliegue.

2.2. Abstracciones por Predicados

Enabledness-Preserving Abstractions. El interés central de este trabajo es utilizar abstracciones acerca del comportamiento de programas para validar que cumplan con su comportamiento esperado. Para eso se suelen crear modelos que luego serán usados para guiar a los desarrolladores, usuarios o auditores a entender el programa y a detectar posibles vulnerabilidades.

Una manera de obtener modelos abstractos de comportamiento es utilizando abstracciones por predicado. Este enfoque consiste en definir un conjunto de predicados y agrupar los estados concretos del programa de acuerdo con la validez de estos predicados. Es decir, cada estado abstracto representa un conjunto de estados concretos que da la misma valuación a todos los predicados.

En este contexto, se introducen las enabledness-preserving abstractions[8], que son máquinas de estados que describen el comportamiento de la implementación de un programa, donde se agrupan los estados en base a qué métodos están habilitados y cuáles no.

Formalmente, se define una clase C como una estructura $\langle M, F, R, inv, init\rangle$ donde $M = \{m_1, ..., m_k\}$ es un conjunto finito de métodos públicos, F es un conjunto de implementaciones de esos métodos, R es un conjunto de precondiciones, inv es el invariante de la clase e $init$ denota las condiciones iniciales, dadas por el constructor. Los conjuntos F y R están indexados por M, es decir, para cada método m_i hay una implementación f_i y una precondición r_i.

Dada una clase C y dos instancias de la misma, c_1 y c_2, se dice que c_1 y c_2 son equivalentes (*enabledness equivalent*, $\equiv_e$) si para todo método $m \in M$, c_1 satisface R_m si y sólo si c_2 satisface R_m.

Cuando se separa a las instancias usando $\equiv_e$ se obtiene un conjunto de estados abstractos, en el que cada uno está mapeado a un conjunto (único) de métodos permitidos. Cada estado abstracto agrupa todas las instancias que comparten el mismo conjunto de métodos permitidos, y puede ser caracterizado por un predicado de estados.

A partir de una clase, se puede definir una EPA como $\langle \Sigma, S, S_0, \delta \rangle$, donde Σ es un conjunto de etiquetas de transición, S es un conjunto de estados abstractos, $S_0 \subseteq S$ es el conjunto de estados iniciales y δ es una función de transición de estados, que establece cómo se conectan los estados por medio de etiquetas. Estos elementos deben cumplir las siguientes condiciones:

- Σ es el conjunto de métodos públicos de la clase C (M).
- S es el conjunto de todos los posibles subconjuntos de M. Cada elemento de S es un conjunto de métodos permitidos, dados por alguna instancia concreta c para la cual además se cumple $inv(c)$. Esto quiere decir que en S están los estados abstractos ya mencionados.
- Los elementos de $S_0 \subseteq S$ son conjuntos de métodos permitidos por alguna instancia c de la clase en la que se satisface tanto el invariante de estado como la condición inicial.
- La función de transición entre estados abstractos δ se define de la siguiente manera: para cada elemento A de S (cada conjunto de métodos permitidos o cada estado abstracto) y para cada método público a de Σ:
  - Si $a \notin A$ (a no está permitido en el estado abstracto A), entonces

    $$\delta(A, a) = \emptyset.$$

  - En cambio, si $a \in A$, definimos a la función de transición como

    $$\delta(A, a) = \{\, B \mid \exists c.\ inv(c) \wedge pred_A(c) \wedge \exists p.\ R_a(c, p) \wedge pred_B(F_a(c, p)) \,\},$$

    donde c es una instancia de la clase, $pred_A$ y $pred_B$ son los predicados que caracterizan a los estados abstractos A y B respectivamente y p es un conjunto de parámetros para el método a. Esta ecuación nos dice que δ va desde A hacia B por medio de la transición a si existe una instancia c del contrato (que cumple con el predicado de A) tal que luego de ejecutar la función a con parámetros p, resulta en una instancia que cumple con el predicado de B.

A modo ilustrativo, la figura 1 presenta la EPA para el contrato inteligente BasicProvenance, cuyo código también se incluye. Este contrato modela un sistema de trazabilidad en una cadena de suministros, manteniendo un registro de las partes involucradas en el mismo y de su estado.

Cada nodo de la EPA representa un estado abstracto y está etiquetado con únicamente las funciones permitidas en ese estado, mientras que cada transición representa una invocación a un método público del contrato.

```
contract BasicProvenance {

    enum StateType { Created, InTransit, Completed }

    StateType State;
    address InitiatingCounterparty;
    address Counterparty;
    address PreviousCounterparty;
    address SupplyChainOwner;
    address SupplyChainObserver;

    constructor(address owner, address observer) {
        InitiatingCounterparty = msg.sender;
        Counterparty = InitiatingCounterparty;
        SupplyChainOwner = owner;
        SupplyChainObserver = observer;
        State = StateType.Created;
    }

    function TransferResponsibility(address newCounterparty) public {
        require(Counterparty == msg.sender);
        require(State != StateType.Completed);

        if (State == StateType.Created)
            State = StateType.InTransit;

        PreviousCounterparty = Counterparty;
        Counterparty = newCounterparty;
    }

    function Complete() public {
        require(SupplyChainOwner == msg.sender);
        require(State != StateType.Completed);
        require(State != StateType.Created);

        State = StateType.Completed;
        PreviousCounterparty = Counterparty;
        Counterparty = address(0);
    }
}
```

Listing 1.1. Código del contrato BasicProvenance

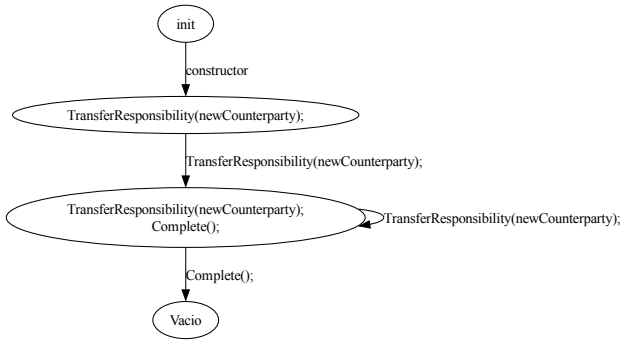

Figura 1. EPA para el contrato BasicProvenance

Otras Abstracciones para Contratos Inteligentes. Además de las EPAs, este trabajo introduce otro tipo de abstracción por predicados, también basada en el estado del contrato en distintos momentos de su ejecución. A estas abstracciones las denominaremos States.

Aquí los estados no representan funciones permitidas y no permitidas, sino que tienen un significado más cercano al entendimiento humano, definido por el programador en base a los predicados elegidos. Los diferentes estados del programa se deciden de forma manual y se pueden introducir predicados arbitrarios para particionar con mayor detalle el conjunto de estados abstractos. De esta manera, se podría hacer más fácil para un desarrollador interpretar la abstracción y así entender en mayor detalle el posible comportamiento del contrato.

Otra posibilidad para generar esta clase de abstracciones es aprovechar las variables que representan explícitamente el estado del contrato mediante enumeraciones (`enum`), como la variable `State` en el contrato BasicProvenance. Este patrón es común en contratos inteligentes, y para identificar, por ejemplo, el estado `Created`, se podría simplemente utilizar el predicado `State == StateType.Created`. Este caso se encuentra ilustrado en la figura 2.

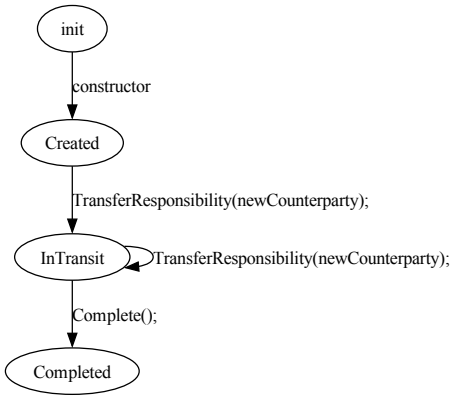

Figura 2. States para el contrato BasicProvenance

2.3. Fuzzing

El *fuzzing* es una técnica para detectar vulnerabilidades que consiste en generar automáticamente entradas aleatorias con el objetivo de provocar errores o comportamientos inesperados en un programa. Un *fuzzer* busca, entre otras cosas, identificar aserciones "violentadas"durante la ejecución. En este trabajo utilizamos Echidna[9], una herramienta de *fuzzing* para contratos inteligentes que

permite ejecutar campañas guiadas por la estructura del contrato y la cobertura de código, con el fin de encontrar violaciones de aserciones que se encuentran dentro del contrato inteligente.

3. Desarrollo del Prototipo

Construcción de las abstracciones. Con el fin de generar estas abstracciones, se desarrolló una herramienta capaz de generarlas automáticamente a partir del código fuente del contrato y de un archivo de configuración. En el caso de querer generar EPAs, dicho archivo proporciona las precondiciones de cada función (información que también podría extraerse automáticamente del código del contrato). En cambio, si se opta por generar una abstracción States, el archivo incluye los predicados de cada estado y una etiqueta asociada a cada uno de ellos. El usuario es el encargado de elegir qué abstracción desea generar y de proveer el archivo de configuración correspondiente.

El método utilizado por la herramienta para generar las abstracciones consiste en incluir dentro del contrato, para cada posible par ordenado de estados y para cada función, una *query* que tiene como objetivo detectar si existe una transición desde un estado hacia el otro ejecutando esa función.

Las *queries* son de la forma presentada en 1.2 y serán ejecutadas permanentemente por el *fuzzer* Echidna junto con el resto de las funciones del contrato, notificando a la herramienta cada vez que un assert es falsificado. En el ejemplo de *query* presentado, en caso de que Echidna detecte que la aserción fue violada, lo notificará y podremos afirmar que desde el estado abstracto `i` mediante la función `function_k` se puede llegar al estado abstracto `j`. Es importante notar que, en caso de que el *fuzzer* no logre falsificar la aserción, no se puede asegurar que esa transición entre estados no existe.

```
function from_i_to_j_by_k() payable public {
    require(predicate_state_i);
    function_k();
    assert(!predicate_state_j);
}
```

Listing 1.2. Ejemplo de query para detectar transiciones entre estados

Una cuestión a tener en cuenta es que en el caso de que se esté construyendo una EPA, el predicado de un estado va a estar determinado por la conjunción de las precondiciones de las funciones habilitadas en ese estado y en el caso States estará determinado manualmente por el usuario y provisto en el archivo de configuración mencionado anteriormente.

Otro aspecto a considerar es que, al generar una EPA, la cantidad de posibles estados abstractos para un contrato con n funciones es 2^n. Esto puede hacer que la técnica sea intratable o excesivamente lenta, afectando la escalabilidad. En la práctica, sin embargo, la cantidad de estados abstractos factibles suele ser considerablemente menor. El prototipo cuenta con dos optimizaciones capaces de reducir la cantidad de estados abstractos posibles sin pérdida de información, lo que a su vez puede reducir considerablemente la cantidad de *queries* generadas.

Integración con herramienta de análisis estático. Aprovechando el trabajo realizado en [10], donde se desarrolló una herramienta con un propósito similar utilizando un verificador formal (VeriSol [11]) en lugar de un fuzzer, se procedió a integrar ambas herramientas con el objetivo de brindar al usuario la posibilidad de seleccionar el tipo de análisis que desea realizar. Esta integración se emplea en la sección de evaluación para comparar ambos enfoques y analizar sus respectivas ventajas y desventajas.

Además, se implementó la funcionalidad que permite utilizar ambas herramientas de manera conjunta, como un acercamiento a la posibilidad de realizar un análisis híbrido aprovechando las fortalezas de cada técnica.

Inclusión de la función `t`. En los contratos inteligentes donde el estado y el comportamiento dependen del valor de `block.number` —comúnmente utilizada para modelar aspectos temporales en Ethereum—, se incorpora al contrato una función denominada `t`, introducida en trabajos anteriores con el fin de simular el avance del número de bloque de manera controlada en contextos distintos a los de una ejecución del contrato en la red real. Para lograrlo, se introduce una variable adicional denominada `blockNumber`, cuyo propósito es mantener un contador interno del número de bloque y se utiliza la función `t()` para incrementar esta variable en una unidad cada vez que se ejecuta, permitiendo emular el paso del tiempo sin depender de la red subyacente.

4. Evaluación

4.1. Metodología de evaluación

A la hora de evaluar el prototipo, se pretende determinar si es posible generar abstracciones por predicados de forma dinámica a partir de contratos inteligentes, si las abstracciones generadas son útiles para comprender el comportamiento de los contratos y para identificar vulnerabilidades, y si se pierde información relevante al usar una herramienta de análisis dinámico como Echidna a diferencia del verificador formal VeriSol.

Para esto, se utilizan dos conjuntos de contratos inteligentes, que ya fueron utilizados en estudios previos. Se trata del *Microsoft Azure Blockchain Workbench* [12] (*benchmark* 1) y del *SmartPulse Evaluation Benchmarks* [13] (*benchmark* 2), que cuentan con 9 y 8 contratos respectivamente.

En cuanto a algunos contratos del *benchmark 1*, estudios previos [14] [15] identificaron vulnerabilidades en su código, las cuales fueron corregidas. Como resultado, surgió una nueva versión de estos contratos, denominada *Fixed*, en la que cada contrato original que tiene un *bug* da lugar a uno nuevo con el mismo nombre seguido de la palabra *Fixed*. Estos contratos también son analizados en este trabajo.

Para el análisis, se ejecutó la herramienta sobre cada uno de los contratos mencionados, utilizando Echidna como *backend*. Se llevó a cabo una ejecución

con el objetivo de obtener la EPA y otra para generar la abstracción de tipo States.

La evaluación se realizó utilizando distintos valores de *test limits*: 1.000, 50.000 y 500.000. El *test limit* es un parámetro de configuración de Echidna que puede interpretarse como el *budget* de la campaña, y representa la cantidad de llamadas a funciones que se realizarán antes de finalizar el proceso de *fuzzing*. La variación de este parámetro tiene como objetivo analizar su impacto en la exploración de los estados del contrato. Es esperable que, a medida que el *test limit* aumente, también lo haga el tiempo de ejecución, pero que, en consecuencia, se logre explorar un mayor número de estados.

Las abstracciones obtenidas se compararon con aquellas generadas al ejecutar la herramienta utilizando únicamente VeriSol como herramienta de fondo, siguiendo la misma técnica empleada en [10], que a su vez contrastaba sus resultados con los de [15].

Se evaluó la capacidad de Echidna para replicar los resultados obtenidos por VeriSol, mediante el análisis de la cantidad de estados y transiciones en las abstracciones generadas. Adicionalmente, ejecutar la herramienta utilizando VeriSol en este contexto nos permite efectuar una comparación acerca de la eficiencia de la herramienta en cada caso, midiendo los tiempos de ejecución. Todas las ejecuciones realizadas utilizando VeriSol como herramienta de fondo se hicieron con un *transaction bound* de 8 y un timeout de 600 segundos, siguiendo las pautas de ejecución utilizadas en [10].

Sumado a esto, en ciertos casos, la abstracción obtenida fue comparada con el modelo conceptual desarrollado por los autores del contrato. Estos modelos están disponibles particularmente para los contratos del *benchmark 1*.

Los contratos, sus archivos de configuración y todos los resultados obtenidos están disponibles, junto con el código de la herramienta, en el repositorio del proyecto [16].

Se fijó, para cada ejecución, un *timeout* total de cuatro horas. Todos los experimentos fueron ejecutados en un equipo con un procesador Intel Core i5 de 3.1 GHz (Dual-Core) y 8 GB de memoria RAM, bajo el sistema operativo macOS 13.7.1.

4.2. Resultados

Viabilidad de la generación de abstracciones mediante *fuzzing*. La totalidad de los resultados puede ser consultada en el repositorio del proyecto[16], en el que también se incluye el código de la herramienta desarrollada y los contratos utilizados.

La gran mayoría de los contratos pudo ser analizada con éxito, obteniendo para cada uno abstracciones de tipo EPA y States. Entre los contratos del *benchmark* 1, hubo dos casos (*Asset Transfer* y *Digital Locker* y sus versiones"*fixed*") en los que no se pudo obtener la abstracción de tipo EPA. Por otro lado, el contrato *ValidatorAuction* del *benchmark* 2 tampoco pudo ser ejecutado con éxito. En estos casos, la ejecución de la herramienta alcanzó el *timeout* antes de completar el análisis, lo que impidió la generación de la abstracción correspondiente.

Esta situación es atribuible a la gran cantidad de *queries* generadas para cada uno de esos contratos, considerablemente superiores al resto, lo que dificulta la exploración de estados por parte de Echidna.

En los demás casos (42 de las 48 ejecuciones totales), las abstracciones se generaron sin inconvenientes. Esto permite afirmar que es posible construir abstracciones por predicados de forma dinámica a partir de contratos inteligentes, utilizando el *fuzzer* Echidna.

Tras analizar detalladamente las abstracciones resultantes, particularmente las de tipo States de los contratos del *benchmark* 1 (de cuyos modelos abstractos esperados disponemos), se puede confirmar que estas permitieron identificar todas las vulnerabilidades esperadas en los contratos, con excepción de *AssetTransfer*. Dichas vulnerabilidades fueron las que motivaron la corrección de los contratos y sus respectivas versiones "*fixed*" en trabajos previos. La ejecución de estas versiones "*fixed*", a excepción de *AssetTransferFixed*, produjo en todos los casos una abstracción idéntica al modelo conceptual provisto por los desarrolladores, hallable en el repositorio[12]. Además, la ejecución de *DigitalLockerFixed* reveló un comportamiento no contemplado en el modelo original.

Esto demuestra la utilidad de las abstracciones generadas por la herramienta, así como su potencial aplicación para comparar el comportamiento real del contrato con el deseado y detectar vulnerabilidades y comportamientos inesperados.

Comparación con VeriSol. Con el objetivo de evaluar la efectividad del enfoque basado en *fuzzing*, se comparan los resultados obtenidos con Echidna con aquellos generados por VeriSol. Se consideran casos exitosos aquellos en los que se identificaron todos los estados y transiciones hallados por VeriSol. Las tablas 1 y 2 resumen qué contratos lograron alcanzar la cobertura deseada —es decir, la alcanzada por VeriSol— para cada tipo de abstracción, utilizando la mejor ejecución de Echidna en cada caso. En las tablas, los casos exitosos se indican con un símbolo de verificación (✓), mientras que los no exitosos se marcan con una cruz (✗). El contrato *ValidatorAuction* no tiene clasificación ya que no pudo ejecutarse ni con Echidna ni con VeriSol dentro del límite de tiempo fijado.

En la mayoría de los casos, que Echidna no encuentre estados o transiciones que VeriSol sí, se lo puede atribuir a una gran cantidad de *queries* generadas y a la especificidad de las precondiciones de las funciones de algunos contratos –como es el caso del contrato *EPXCrowdsale*–, cosa que torna más compleja la exploración del fuzzer. Por otro lado, en el caso del contrato *SimpleAuction*, que su generación de la EPA no haya sido exitosa está vinculado con el no determinismo de Echidna, ya que la única transición faltante con un *test limit* de 500.000 sí fue encontrada al ejecutar la herramienta con un *test limit* de 50.000, aunque otras transiciones sólo se encontraron con el límite mayor.

Nombre del contrato	EPA	States
Asset Transfer	✗	✗
Asset Transfer Fixed	✗	✗
Basic Provenance	✓	✓
Basic Provenance Fixed	✓	✓
Defective Component Counter	✓	✓
Defective Component Counter Fixed	✓	✓
Digital Locker	✗	✓
Digital Locker Fixed	✗	✓
Frequent Flyer Rewards Calculator	✓	✓
Hello Blockchain	✓	✓
Hello Blockchain Fixed	✓	✓
Refrigerated Transportation	✓	✓
Refrigerated Transportation Fixed	✓	✓
Room Thermostat	✓	✓
Simple Marketplace	✓	✓
Simple Marketplace Fixed	✓	✓

Cuadro 1. Cobertura Contratos Benchmark 1

Nombre del contrato	EPA	States
Auction	✓	✓
Crowdfunding	✓	✗
EPXCrowdsale	✗	✗
EscrowVault	✓	✓
RefundEscrow	✓	✓
RockPaperScissors	✓	✓
SimpleAuction	✗	✓
ValidatorAuction	-	-

Cuadro 2. Cobertura Contratos Benchmark 2

Otra métrica relevante para comparar la utilización de Echidna y VeriSol es el tiempo de ejecución.

Al comparar los tiempos de ejecución, se observa que VeriSol obtiene resultados en tiempos similares a los de Echidna con un *test limit* de 50.000. Sin embargo, para evaluar la eficiencia de cada herramienta, es necesario analizar también las abstracciones generadas.

Por ejemplo, en contratos pequeños como DefectiveComponentCounter, HelloBlockchain y SimpleMarketplaceFixed, todos pertenecientes al benchmark 1 y con menos de 25 queries generadas cada uno, un *test limit* de 1.000 produce la misma abstracción que VeriSol, incluso con un menor tiempo de ejecución.

En cambio, en contratos con una mayor cantidad de queries, como Auction o Crowdfunding en modo EPA (ambos con 96 queries), únicamente con un *test limit* de 500.000 se encuentran todos los estados y transiciones, pero con un tiempo de ejecución más de cinco veces mayor al de VeriSol.

Los resultados temporales para estos cinco contratos se pueden encontrar en la figura 3. En la misma, #Q representa la cantidad de *queries* generadas para cada contrato, TL indica el *test limit* correspondiente a la ejecución de Echidna

y TXB el *transaction bound* utilizado al correr VeriSol. El resto de los resultados se encuentra disponible públicamente ([16]).

Nombre del contrato	#Q	TL=1k	TL=50k	TL=500k	TXB=8
DefectiveComponentCounter	2	3.94	30	260.11	7.26
HelloBlockchain	2	3.72	24.59	209.7	7.18
SimpleMarketplaceFixed	24	4.31	24.52	200.22	19.13
Auction	96	17.45	57.73	373.13	54.16
Crowdfunding	96	26.14	59.55	387.32	54.63

Cuadro 3. Tiempo (s) - Echidna vs. VeriSol

Cuando Echidna se ejecuta con un *test limit* de 500.000, el tiempo de análisis aumenta considerablemente respecto al resto de las ejecuciones. Esto indica que, si bien límites mayores permiten descubrir más transiciones, el costo computacional asociado es significativamente superior al de VeriSol (excepto en un caso, todas las ejecuciones de Echidna con un *test limit* de 500.000 demoraron más que VeriSol con un *transaction bound* de 8).

En términos generales, VeriSol es más eficiente, especialmente cuando el *test limit* de Echidna es alto. No obstante, para contratos con una baja cantidad de estados abstractos por detectar, podría ser viable ejecutar Echidna con un *test limit* bajo. Esto sugiere la posible implementación de una técnica que estime un *test limit* adecuado en función del tamaño del contrato de entrada y la cantidad de queries generadas para el mismo, entre otros factores.

5. Conclusiones y trabajo futuro

Este trabajo exploró la generación y el uso de abstracciones por predicados para analizar el comportamiento de contratos inteligentes de Ethereum y detectar posibles vulnerabilidades, en particular aquellas relacionadas con la lógica de negocio, donde puede existir una discrepancia entre los requerimientos esperados y la implementación concreta del contrato.

Se introdujeron formalmente las *enabledness-preserving abstractions* (EPAs) y se analizaron distintos tipos de abstracciones ya aplicadas en trabajos previos sobre contratos inteligentes.

A su vez, se desarrolló el primer prototipo que emplea análisis dinámico para construir automáticamente abstracciones por predicados para contratos inteligentes escritos en Solidity, particularmente usando el *fuzzer* Echidna.

Mediante la evaluación de la herramienta sobre un conjunto de contratos inteligentes, se demostró que es factible generar abstracciones de manera dinámica con Echidna y se realizó una comparación con el uso del analizador estático VeriSol, estudiado en un trabajo previo [10]. Los resultados muestran que ambas herramientas generan abstracciones similares en la mayoría de los casos, aunque el uso de Echidna conlleva un tiempo de ejecución considerablemente mayor, especialmente en los contratos de mayor tamaño. Además, se observó una esperable

cuota de no-determinismo en Echidna, lo que implica que diferentes ejecuciones pueden producir resultados distintos, incluso con los mismos o mayores recursos.

Por otra parte, se analizó el impacto del *budget* (*test limit*) en el rendimiento y cobertura de la herramienta, concluyendo que un mayor budget incrementa la cantidad de estados y transiciones exploradas, aunque también aumenta el tiempo de ejecución de manera significativa. También se mencionó una posible limitación de Echidna en la generación de valores para alcanzar ciertos estados cuyas precondiciones cuentan con múltiples restricciones sobre diferentes variables, lo cual refleja una limitación inherente del *fuzzing* en general para satisfacer precondiciones complejas o altamente restrictivas.

A modo de conclusión, la herramienta desarrollada representa un aporte valioso para desarrolladores y auditores de contratos inteligentes, al permitir la generación automática de abstracciones útiles mediante análisis dinámico.

Entre las líneas de trabajo futuro, se destaca la necesidad de abordar las limitaciones de escalabilidad observadas, especialmente en aquellos contratos que no pudieron ser analizados con éxito dentro de los límites de tiempo establecidos. Una posible mejora consiste en incorporar pasos previos a la ejecución, que permitan reducir el espacio de búsqueda de la herramienta. En este sentido, una opción es la utilización de *SMT solvers* para detectar y descartar automáticamente estados abstractos inherentemente inalcanzables. Por ejemplo, si un estado está asociado al predicado $A \wedge B \wedge \neg A$, la herramienta debería identificar esta contradicción lógica y eliminar dicho estado antes de intentar generar la abstracción.

Otra alternativa consiste en dividir las *queries* generadas en múltiples contratos más pequeños, en lugar de integrarlas todas en un único contrato. Esta fragmentación podría reducir el tiempo de ejecución (al trabajar sobre contratos más simples) y mejorar la cobertura del análisis. Al haber menos funciones por ejecutar, cada función se ejecuta más veces con distintos parámetros, lo que beneficia la exploración de estados concretos del contrato bajo prueba.

Adicionalmente, se plantea la posibilidad de integrar técnicas de ejecución simbólica, que permitan inferir automáticamente las restricciones que deben cumplir los parámetros de entrada para que una función sea ejecutable. A diferencia del *fuzzing* sin restricciones, la ejecución simbólica puede analizar el código y deducir los valores de los parámetros para los cuales una función puede ejecutarse exitosamente, sin ser revertida automáticamente.

Por último, cabe señalar que los contratos utilizados en este trabajo fueron diseñados para evaluar el rendimiento de herramientas de análisis, y no necesariamente se corresponden con contratos utilizados en entornos reales. Por ello, sería relevante evaluar la herramienta sobre contratos inteligentes reales, ya sean de código abierto o en producción, a fin de analizar su efectividad en escenarios prácticos y más representativos.

Referencias

[1] Zheng, Z., Xie, S., Dai, H., Chen, X., & Wang, H. (2017). An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. *2017 IEEE International Congress on Big Data (BigData Congress)*, 557-564. https://api.semanticscholar.org/CorpusID:29591273

[2] Buterin, V. (2014). Ethereum: A next-generation smart contract and decentralized application platform. https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf

[3] Nakamoto, S. (2009). Bitcoin: A Peer-to-Peer Electronic Cash System. *Cryptography Mailing list at https://metzdowd.com*.

[4] Ethereum Foundation. (n.d.). *Ethereum Virtual Machine*. https://ethereum.org/en/developers/docs/evm/

[5] Solidity Team. (s.f.). *Solidity language repository*. https://github.com/ethereum/solidity

[6] Solidity Team. (s.f.). *Solidity language portal*. https://soliditylang.org

[7] Solidity Team. (s.f.). *Solidity compiler versions*. https://soliditylang.org/blog/category/releases

[8] de Caso, G., Braberman, V., Garbervetsky, D., & Uchitel, S. (2012). Automated Abstractions for Contract Validation. *IEEE Transactions on Software Engineering*, *38*(1), 141-162. https://doi.org/10.1109/TSE.2010.98

[9] Trail of Bits. (2018). Echidna: A Fast Smart Contract Fuzzer. https://github.com/crytic/echidna

[10] Torres, E. (2023). Generador de abstracciones para smart contracts. Tesis de Licenciatura en Ciencias de la Computación.

[11] Microsoft Research. (2019). *VeriSol: A formal verifier for Solidity based smart contracts*. https://www.microsoft.com/en-us/research/project/verisol-a-formal-verifier-for-solidity-based-smart-contracts/publications/

[12] Microsoft Corporation. (2020). Azure Blockchain Workbench. https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples

[13] Utopia Research Group. (2021). SmartPulse Evaluation Benchmarks. https://github.com/utopia-group/SmartPulseTool/tree/master/benchmarks/evalBenchmarks

[14] Antonino, P., & Roscoe, A. W. (2021). Solidifier: bounded model checking solidity using lazy contract deployment and precise memory modelling. *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 1788-1797. https://doi.org/10.1145/3412841.3442051

[15] Godoy, J., Galeotti, J. P., Garbervetsky, D., & Uchitel, S. (2022). Predicate abstractions for smart contract validation. *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, 289-299. https://doi.org/10.1145/3550355.3552462

[16] Grinspan, I. (2025). Dynamic Smart Contract Abstractor - Herramienta y resultados. https://github.com/igrinspan/dynamic-smart-contract-abstractor